

Improving Disk Cache Hit-Ratios Through Cache Partitioning

Dominique Thiébaut, *Member, IEEE*, Harold S. Stone, *Fellow, IEEE*, and Joel L. Wolf

Abstract—This paper presents an adaptive algorithm for managing fully associative cache memories shared by several identifiable processes. The on-line algorithm extends an earlier model due to Stone, Wolf, and Turek, and partitions the cache storage in disjoint blocks whose sizes are determined by the locality of the processes accessing the cache. Simulation results of traces for 32-megabyte disk caches show a relative improvement in the overall and read hit-ratios in the range of 1% to 2% over those generated by a conventional Least Recently Used replacement algorithm. The analysis of a queueing network model shows that such an increase in hit-ratio in a system with a heavy throughput of I/O requests can provide a significant decrease in disk response time.

Index Terms—Disk cache memories, memory hierarchy, multitasking, optimal partitioning algorithm, performance measurement, performance prediction, program behavior.

I. INTRODUCTION

As processor speeds continue to increase, the relative penalty for access time to disk becomes a major source of performance degradation. To overcome this problem, many disk systems have been designed with a large semiconductor random-access memory acting as a cache. However, the characteristics of such memories are somewhat different from the caches associated with processors. High-speed processors require cache access times to be in the low nanoseconds, which precludes the ability to perform nontrivial processing during a cache search or between cache accesses. In contrast, a great deal of processing can be performed during and between searches of disk caches. This enables a disk cache to be managed by more sophisticated and complex techniques than processor caches. Moreover, the very lengthy access time to rotating mechanical memories means that improvements in cache management that result in lower miss rates are of very high benefits.

In this paper we address the problem of how to manage disk caches to achieve high performance. As a baseline we use fully associative searches and least recently used (LRU) replacement with no prefetching. We investigate the improvement attainable by partitioning the cache into regions of data that share some similar characteristic, with each of the regions managed by LRU replacement. At a higher level, the cache management algorithm determines the size of each region in the partition,

Manuscript received January 12, 1990; revised January 22, 1991. This work was supported in part by IBM, under Contract 17870015.

D. Thiébaut is with the Department of Computer Science, Smith College, Northampton, MA 01063.

H. S. Stone and J. L. Wolf are with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9105059.

and alters the size dynamically in response to changes in the access pattern. Both the base LRU algorithm and the partitioned LRU algorithm are of the type known as *load-driven* [6]. The partitioned LRU algorithm has the property that, given a fixed-size cache, the cache is divided into regions whose sizes are selected such that the hit-ratio to the cache as a whole is maximized. Although we focus on disk caches in the validation section, our algorithm is independent of the implementation and can be applied to any buffering system managed by an LRU algorithm that has sufficient processing capability to perform the required operations in real-time.

Earlier related work on disk-cache replacement algorithms includes a paper by Bentley and McGeoch [2] who used real data to compare and rank several replacement algorithms for disk caches. Their data show that conventional LRU is superior to other replacement algorithms, such as *frequency count*, *FIFO*, or *transpose*. Their result is proven for the general cache by Sleator and Tarjan [22].

As Smith points out [14], hit-ratio is one of several performance measure of a disk-cache, as the cache performance is a function not only of the hit ratio, but also of such parameters as the number of disks, number of controllers, the disk-access method, the interconnection between the I/O channel and the disk controllers, or the varying amount of prefetching incurred on each access (fetch until end-of-track, for example). Because such details vary greatly from one system to another, because they are immensely influenced by the technology, and since more accurate measures such as the response time behave as monotone functions of the hit-ratio, the cache hit-ratio, nonetheless, provides valuable information that can be input to more sophisticated models. In Section III we analyze the influence of small increases in hit-ratio on the response time of a typical 32-disk computer system, and learn that in congested systems a small increase in hit-ratio can provide significant decrease in I/O response time.

Our model is a direct extension of the model presented by Stone, Wolf, and Turek [15], who show that when two Processes *A* and *B* share a cache of size *C*, partitioning the cache by allocating *C_A* lines to Process *A*, and *C_B* lines to Process *B* (*C_A + C_B = C*) maximizes the hit ratio when the miss-rate derivative of Process *A* as a function of cache size, in a cache of size *C_A* is the same as the miss-rate derivative of Process *B* in a cache of size *C_B*. In this paper we use the term *partition* in its mathematical sense, that is, all subsets are disjoint, and their union forms a set whose cardinality is the capacity of the cache. The results of Stone, Wolf, and Turek hold for both set-associative caches and for fully associative

caches, but we use them in this paper only in the context of fully associative caches.

The paper presents a practical implementation of the replacement policy of Stone, Wolf, and Turek and provides a new means for deriving an approximation of the miss-rate derivative on-line. The model is adaptive and requires the use of a shadow directory [12] for each class. Since the shadow directory contains only the block addresses and no data, the increased storage required by the algorithm is fairly small. For example, assuming a cache containing 8192 blocks of 4096 bytes each, where the entry for each shadow directory contains a 16-bit tag and two 16-bit counters, the implementation of the partitioning algorithm requires only 1.9% more storage than a one-class, fully associative cache.

We evaluated the partitioning model by a trace-driven simulation based on a trace of 1.6 million disk I/O accesses directed to 13 physical disks sharing one cache and its associated cache controller. The study reveals that the partitioning algorithm performs slightly better on 32-megabyte caches than the base LRU algorithm (where all items are considered to be of the same class), with absolute increases in hit-ratio of 1% to 2%. The partitioning algorithm performs better on smaller caches with absolute increases running roughly 2% to 5.6%. The relative performance improvement depends on several factors including the hit ratio and the throughput of disk accesses generated by the processor. A queueing network model of a simple configuration with 32 disks showed that an increase in disk read hit-ratio of 2.5%, over a wide range of absolute hit-ratios, results in average decreases in disk response time varying from 2.78% for lightly loaded systems, to 20.22% for heavily utilized systems.

In the simulation of the 32-megabyte caches, when write operations are counted as misses (*write-through*), the read hit-ratios of the partitioned caches are only slightly higher than the read hit-ratios of the equivalent nonpartitioned caches. For smaller caches, the partitioning algorithm produces better improvements, but they are still only a few percent higher in absolute value. The model presented here can easily be modified to select the optimum partition as a function of some other measure directly related to performance than the hit ratio.

Partitioning memory among different processes or tasks has been studied extensively in the context of paging in a virtual memory system. Although the goals are different from those associated with disk caching, it is interesting to look at different approaches, to study the selection of the objective functions and performance measures, and to examine the results achieved.

Chu and Opderbeck [5] introduce the page-fault frequency replacement algorithm which measures the number of page faults experienced over a given interval of time, and compares it to some threshold. A page-fault frequency below the threshold indicates that too much memory is allocated to the process, and that process's storage is allowed to shrink. Conversely, a high page-fault frequency indicates that the process could benefit by being allocated more memory, and the storage allocation is allowed to grow. Chu and Opderbeck present arguments for implementing such algorithm in a multiprogramming environment, but do not present any data.

Chamberlin, Fuller, and Liu [4] propose an interesting heuristic approach which yields near, but not optimal, performance when programs are characterized by their *lifetime function*. The heuristic is to allocate space for each process such that their page-fault rates become identical. The measure of performance for optimality is defined as a complex function involving a combination of such parameters as the lifetime function, wait time to service a page, and processor speed. Chamberlin *et al.* show that the heuristic generates a partition that achieves from 87 to 92.5% of the optimal measure.

Denning *et al.* [6] consider three approaches to the problem of partitioning the pages of multiprogrammed tasks in a virtual memory. The *knee criterion* is presented as the most robust of the three, and arguments are presented (without formal proofs) that partitioning the memory such that each process is allocated enough storage to contain the knee of its lifetime function [1] "tends to minimize the component of memory space-time due to paging." The experiments reported show that the knee criterion always yields a system throughput within 5% of optimal throughput, although the practical on-line measure of the lifetime function appears to be a nontrivial task.

Another attempt at using the individual process's page-fault rate to drive the partitioning algorithm is that of Brandwajn and Hernandez [3], who follow the *program-driven* approach [6]. In their approach, the *load*, defined as the number of processes sharing the memory, is determined as the number of working sets that can be placed together in memory. Their method works well when all processes have identical or near-identical fault-rates, but severely deteriorates when different page-fault rates are considered.

Although the goal and domain of application of these research efforts is on paging, which is different from caching disk-sectors or tracks of data, they share with the problem we address the goal of allocating storage to different processes with varying levels of locality, such that some performance measure is maximized. All show that measuring process locality is very hard to accomplish on-line, and furthermore that, in most cases, the performance obtained is close to, but not equal to, the optimum.

In the next section we derive the partitioning algorithm and present the different assumptions and approximations used in its formulation. Section III presents a trace-driven simulation of fully associative 4-megabyte to 32-megabyte caches, and an analysis of the improvement resulting from the partitioning algorithm. In Section IV we discuss the application and possible enhancements of the partitioning algorithm. We conclude the paper in Section V by listing open research problems.

II. THE PARTITIONING ALGORITHM

Fig. 1 illustrates the partitioning model of Stone, Wolf, and Turek (SWT). Two processes, Process A and Process B, are competing for the same 1024-line cache. The miss rate of Process A is shown in the upper part of the graph, for increasing cache sizes. The miss rate of Process B is shown on the same graph, for decreasing cache sizes. Assuming the lines of Process A are stored on the left-hand side of the cache, while the lines belonging to Process B are stored

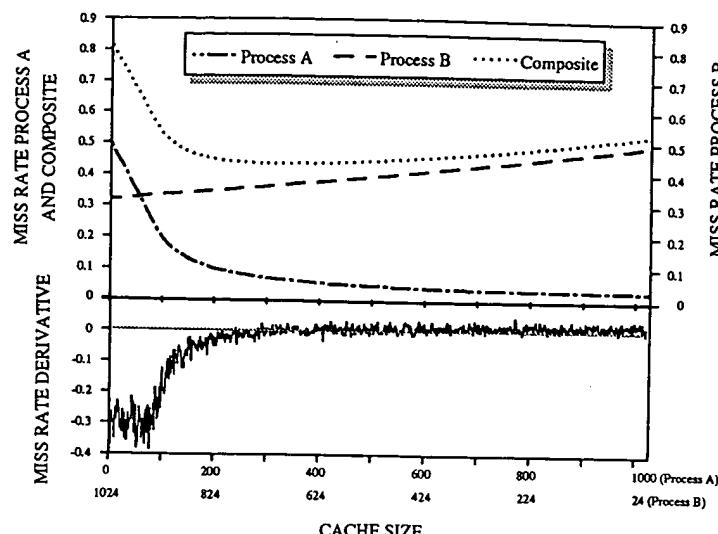


Fig. 1. Miss rates and composite derivative for two processes.

on the right side of the cache, and that the cache is filled with lines from the two processes, then the composite miss rate of the two processes is computed very simply. First a vertical line dividing the cache into a partition of *A* and *B* is drawn. The intercepts of this line with the miss-rate curves corresponds to the miss rates experienced by the *A* and *B* processes. The composite miss-rate is simply the sum of these two terms. Obviously the optimum partition corresponds to dividing the cache so that the vertical line intersects the composite miss rate at its global minimum. The SWT model shows that when the optimum partition is reached, the derivatives of the miss rates of Process *A* and Process *B* with respect to cache size are identical. If references by Process *A* and Process *B* occur at unequal frequencies, the optimum partition occurs at the point where the frequency-weighted miss-rate derivatives are equal.

Extending the SWT model to more than two processes sharing the cache is a typical example of a *resource allocation problem* (RAP) [7], [18]–[20], and has been shown by Ghanem [10].

The disk cache explored in this paper is shared by 13 physical disks, offering two possible partitioning schemes.

- 1) Partitioning based on the physical property of the disk system. A natural way to use the partitioning model for our purposes is to associate a distinct process with each physical disk. The cache-partitioning problem becomes a problem in partitioning a large cache among 13 competing processes, each associated with a distinct disk.
- 2) Partitioning based on the file system. In this scheme, one partitions the composite stream of disk accesses into streams associated with operating system, other systems programs, and individual application programs. These streams are likely to exhibit characteristic patterns that can be used to determine how the cache should be partitioned among the processes.

We shall concentrate on the first scheme in this paper, as we did not have sufficient information in the trace to facilitate the second type of partitioning.

A. Approximating the Miss-Rate Derivative

We now discuss the practical implementation of the SWT model by providing an on-line algorithm to compute a close approximation of the miss-rate derivative. The miss-rate of a process in a cache of size *i* can be represented as the percentage of lines that flow through the *i*th cell of an infinite LRU stack, as shown in Fig. 2. Indeed, when a miss occurs in a cache of size *i*, a new line is brought to the most-recently used (MRU) position of the stack, and all the other lines shift to the right. Hence, the line at position *i* in a cache of size *i* moves to the right on each miss experienced by the cache. The miss-rate derivative represents the incremental change in this flow as the boundary of the *i*th cell is moved by a small amount δ . Of course, since the cache size is discrete, the miss rate is not a continuous function of the cache size, and its derivative is not defined. We can, however, generate a smooth continuous curve joining the discrete miss-rate values and compute the derivative of that smooth curve at the points of interest. This is the source of our first approximation. The second approximation is to set δ to 1.

The miss rate of a process as used here is the instantaneous rate of misses per reference. In this case, the miss-rate derivative is the difference of the flows into and out of a cell of the cache. We approximate the miss rate of a process by computing the ratio of misses to references. Then the derivative of the miss rate with respect to cache size for a cache of size *i* is estimated as

Miss-rate derivative

$$\approx \frac{(\text{Number of } A\text{-misses in } C_i)}{\text{Total number of } A \text{ and } B \text{ references}}$$

$$- \frac{(\text{Number of } A\text{-misses in } C_{i+1})}{\text{Total number of } A \text{ and } B \text{ references}}$$

$$= \frac{(\text{Number of } A\text{-misses in } C_i - \text{Number of } A\text{-misses in } C_{i+1})}{\text{Total number of } A \text{ and } B \text{ references}}$$

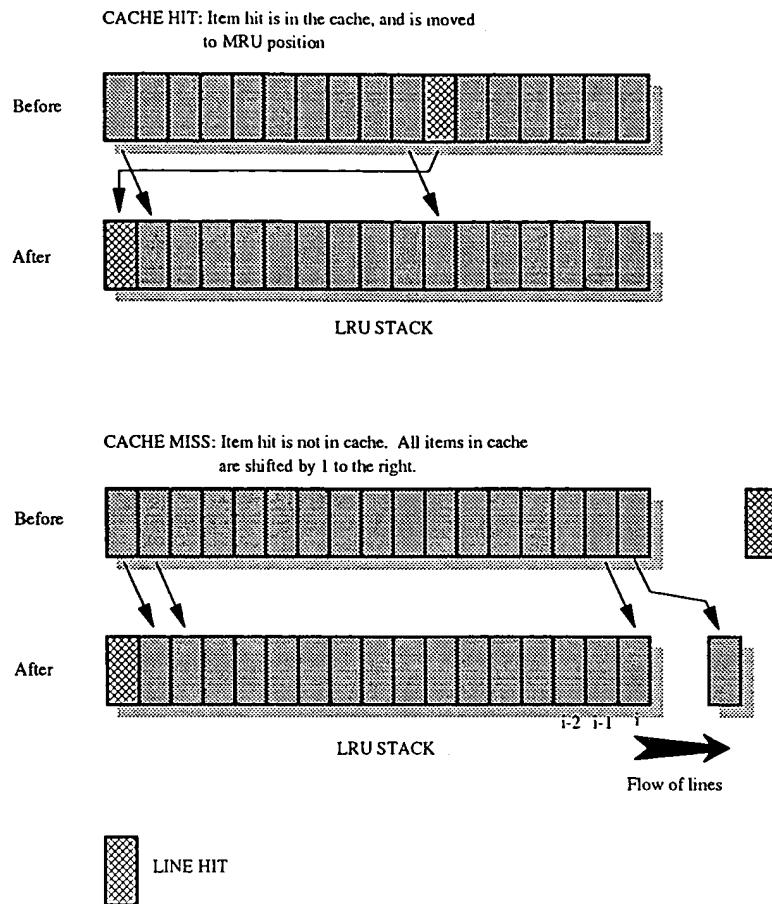


Fig. 2. Estimation of miss-rate derivative.

where C_i denotes a cache of size i . For our purposes, we are interested in the miss rate derivative of the subsequence of requests belonging to a specific process intermingled with request from other processes.

Since the miss rate generated above varies with the number of accesses to A and B , it represents the *weighted* miss-rate of a process, where the weight is the frequency of access of one process relative to the total amount of accesses. The numerator in the last fraction above is equal to the number of misses in a cache of size i that become hits in a cache of size $i + 1$. The weighted miss-rate derivative for Process B is defined similarly.

Both the miss-rate derivative for Process A and the miss-rate derivative for Process B have the same denominator. As a result, partitioning the cache where the miss-rate derivatives are the same can be accomplished by partitioning at the point where the numerators of the above expressions relating to Process A and B are equal. The numerator represents the number of hits at index $i + 1$ in the LRU stack. We refer to the collection of this quantity over the range of cache sizes C_i as the *histogram of hit indexes* or *hit-index distribution*. Our partitioning scheme divides the cache at the point where the histograms of hit indexes for all processes have identical height. This method of partitioning where the hit index histograms have the same values has the convenience of automatically weighting the derivatives by frequency of ref-

erence. For example, assume that two processes with identical statistics share the cache. If the cache receives equal amounts of requests from both processes, then the algorithm partitions the cache in equal halves, since both processes have identical miss-rate curves. On the other hand, if one process accesses the cache less frequently than the other, its hit-index histogram is lower in amplitude than that of its counterpart. Comparing the hit-index distributions favors the process with the heaviest use of the cache, which is consistent with intuition.

B. Problems with Nonmonotonic Miss-Rate Derivatives

Finding two bars in two different histograms with (almost) identical heights reduces to a trivial search problem if the histograms are monotonic. Unfortunately, the histograms of hit indexes are not generally monotonic, as Fig. 3 illustrates.

In this figure we plotted the typical shape of miss-rate curves observed in fully associative caches by using two synthetically generated traces [17]. Note that for caches of size 1, 2, 3, and 4, the miss rate is very close to 100%, and starts decaying hyperbolically for caches larger than 5. This distinct behavior on each side of the knee has been observed and modeled in Thiébaut [16] and is a consequence of the inability of a program to load its working set into small caches, yielding a very high miss rate that changes slowly with cache size until the cache is large enough to hold a complete local context. As cache size increases beyond this point, the miss rate diminishes

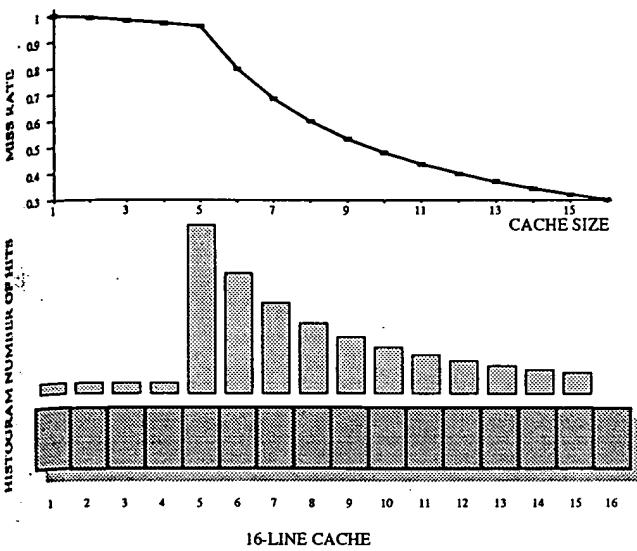


Fig. 3. Miss rate and hit-index histogram in 16-line cache.

more rapidly with cache size, and a knee forms in the miss-rate curve.

Since the miss-rate curve is not continuous, its derivative exhibits a discontinuity at the knee. Because the hit-index distribution is an approximation of the weighted miss-rate derivative, it changes abruptly with cache size around the knee.¹ The miss-rate curve is shown in Fig. 3. As a result, it is possible for the algorithm in charge of finding the optimal partition by searching the histogram of the hit indexes to "lock in" the small well of the curve that appears in Fig. 3. At least two approaches to solve this problem are possible. The first one is to implement a search algorithm that takes into account the possibility that the well exists, and to avoid locking up in the well. The second is to smooth the histogram into a monotonic function before undertaking the search.

Sorting the histogram by decreasing magnitudes turns out to be a fast, inexpensive, and accurate smoothing process, as indicated by Fig. 4. The hit-index distribution shown in this figure is that of a sequence of disk I/O requests that were fed to a fully associative cache simulator. The hits to the first 1200 cells of the cache are shown. Note that the sorted distribution is monotone and provides a faithful representation of the true distribution.

C. Updating the Partition: The "Robin Hood" Philosophy

At this point we describe the adaptive process that our algorithm must perform to maintain a partition that optimizes the hit ratio. This adaptive process must detect changes in the characteristics of the processes accessing the cache, and must respond by modifying the partition. We first describe the two-partition case in which the algorithm maintains a cache shared by two processes, *A* and *B*, and then generalize to the *N*-partition case.

¹Miss-rate curves published in the literature do not always have such knees. A possible reason for a missing knee is that the range of cache sizes simulated may exclude the regions where the cache is small enough to play such a characteristic.

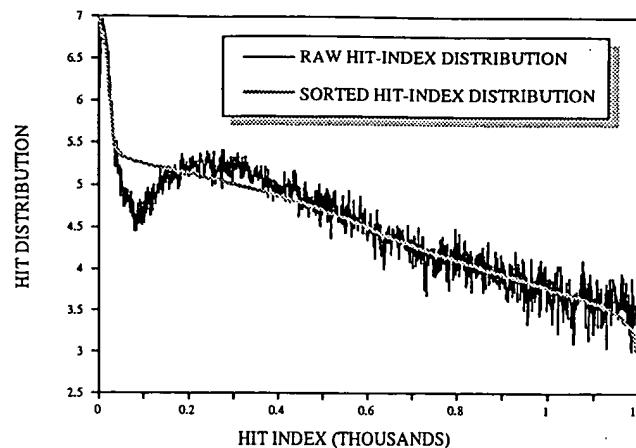


Fig. 4. Smoothing hit distribution by sorting.

We know from the previous sections the conditions that our two-process partition should satisfy. Given that the distribution of hit-indexes changes with every cache hit, how do we reach the optimum partition, and how do we maintain it? Our answer is to adopt the "Robin Hood" philosophy: take from the rich and give to the poor.

Assume that Process *A* has just experienced a miss in the cache. We check the value of its *smoothed* hit distribution at the index equal to its current partition size, and we compare it to the smoothed hit-distribution of Process *B* at its partition index. If the value read for *A* is larger than that read for *B*, then the frequency-weighted miss-rate derivative of *A* is larger (in absolute value) than the frequency-weighted miss-rate derivative of Process *B*. In other words, Process *A* would see a greater decrease in miss rate if we were to allocate one more cache block to its partition than Process *B* would. *B* is the "rich" process, while *A* is the "poor" one. In this case we increase *A*'s partition by 1 block and decrease *B*'s by 1. When decreasing *B*'s partition, we remove its LRU line. Hits do not alter the partition size, but affect the hit-index distribution.

Because processes may change characteristics over time, the algorithm must also attribute different weights to the information that it records about the processes, so that more recent behavior outweighs the older behavior. For example, a process may access the cache heavily for a brief period of time, leading the algorithm to allocate that process a significant share of the cache. Then the process may suddenly enter a dormant period. In such a situation, the algorithm must slowly reduce that process's partition to favor the processes that are still active. Since the dormant process does not update its hit index distribution, we need a mechanism to measure the history of a process and to combine such history with the histogram. We accomplish this by operating the algorithm in a time window of length τ . Every τ accesses to the cache, the hit distribution of each process is divided by some constant C_τ . As a result, assuming that t is the current virtual time, and that t is a multiple of τ , hits that occurred in the widow $[t - \tau, t]$ contribute to the histogram with a weight of 1, hits that occurred in the time window $[t - 2\tau, t - \tau]$ contribute with a weight of $1/C_\tau$, hits that occurred in the time window $[t - 3\tau, t - 2\tau]$ contribute with a weight of $1/C_\tau^2$, etc. Thus,

the past intervals are weighted by factors that diminish by powers of C_τ .

We now briefly investigate what parameters affect the selection of the parameters controlling this exponential smoothing. During the window τ , the partitioning algorithm obeys fixed rules to determine which process is allowed to grow or shrink in the cache. At the transition between a window and the next one, the rules are updated using the data collected during the first window.

Several questions face the designer: Should the window τ be left constant over time, or should it be allowed to shrink and grow as the processes exhibit different locality properties? If the window is adjustable, should it be a common window controlling the update of all processes, or should each Process P_i be associated its own window τ_i ? In all cases, what is the relationship between τ and C_τ ?

Although an in-depth analysis of the behavior of the partitioning algorithm as a function of τ and C_τ is beyond the scope of this paper, we explore some possible solutions, based on the work of Fagin.

Fagin [9] studies the relationship between the asymptotic miss-rate of a storage managed by an LRU replacement-policy, and by a *working-set* (WS) replacement policy when the blocks referenced exhibit a probability distribution obeying a Zipf law [21], [11], better known for one of its member functions, the 80/20% rule. Here, Fagin assumes that the references to blocks are independent, obeying the independent reference model (IRM), but also that some blocks are "hotter" than others, and are referenced more often. Assuming a total of N blocks, the probability of accessing Block b_i is defined as

$$\Pr[i] = z_i = ki^{-\alpha} \quad (1)$$

where k is the normalization constant assuring that the sum of all $\Pr[i]$ adds up to 1, and α is defined as the *skewness* of the block accesses. The 80/20% rule corresponds to $\alpha = 0.86$. Other values of α in the range 0 to 1 yield different proportions.

Fagin shows that, as the number of blocks N becomes very large, the miss-rate of the LRU and WS-managed storage converge toward the same asymptotic values, and he gives equations relating the window size over which the miss-rate is computed to the size of the storage. In other words, given a window-size τ_0 , a skewness α , and a target miss-rate μ_0 , one can find the storage size C_0 for which the LRU or WS miss-rate will be equal to μ_0 .

One can then conceive a simple scheme where the disk-cache controller selects an original storage size $C_0^{(i)}$ and a window size $\tau_0^{(i)}$ for Process P_i . At the end of the window, estimates of $\mu_0^{(i)}$ and $\alpha^{(i)}$ are computed. As time evolves, the controller can detect sudden changes in $\alpha^{(i)}$ as the partition $C_0^{(i)}$ changes, and can adapt the window size $\tau_0^{(i)}$ accordingly to produce a minimal miss rate $\mu_0^{(i)}$.

Another interesting approach worth noting is that of assigning individual (τ, C_τ) pairs to each process competing for the cache, each pair varying independently as the associated process evolves. One could imagine selecting a short τ window when the process just starts, so as to quickly adapt to that

process' locality properties, and lengthening the window as the process evolves in a steady state. A sudden prolonged burst of misses could be used to trigger a shrinking of the window, and/or an increase in C_τ to allow the algorithm to track the changes more closely.

D. The Two-Process Model

In this section we describe the details of the partitioning algorithm applied to two processes.

Because we need to compute the hit-index distribution for both processes, we need to maintain a shadow directory [12] for each process. The shadow directory is simply the part of the cache that contains the addresses of the blocks stored in the data array. Since the size of a cache block may be several orders of magnitude larger than the size of the tag identifying it,² the storage cost incurred in maintaining several shadow directories is relatively small. With each shadow directory is associated two arrays, `hit_index`, and `smoothed_index`. The first array records the number of hits to each entry in the directory, and contains the raw hit distribution. The second array contains the distribution of hit-indexes sorted by number of accesses (smoothed), and is updated at the end of each window of time.

For the 32-megabyte disk-cache simulation results presented later, we elected to update the smoothed distribution from the raw one every $\tau = 10\,000$ references for track caches, and every 100 000 references for sector caches.

We applied the partitioning algorithm to the two synthetic traces used in Fig. 1 and recorded the variation of the partition of the two processes. The variation of the partition is shown in Fig. 5. The evolution of the partition and the corresponding miss-ratio as a function of time is shown as the *walk* of a point of coordinates (x, y) , where x is the amount of cache allocated to Process A , and y is the overall miss ratio of the partitioned cache. Horizontal movements of the point indicate a change in partition. A downward move of the points indicates that the miss-ratio is decreasing. If the algorithm works correctly, we expect the walk taken by the point to move downwards to an area where the miss-ratio is minimum, i.e., toward some "well" of minimum miss-ratio.

This is indeed what is observed. The beginning of the walk, as well as its last location are flagged by two arrows. Note that the walk terminates in the region where the composite miss-rate is fairly flat and close to its minimum.

E. Partitioning Cache for Several Competing Processes

Extending the partitioning model from two to P processes ($P > 2$) is trivial. Every time a Process P_j misses in the partitioned cache, we find the Process P_j with the smallest `SmoothIndex[PartitionPj]` value (the richest process). If this process is different from P_i (the poorer process), we decrease the partition of P_j and increase that of P_i .

By continuously taking away from the richest process, the algorithm brings that process closer to the pack of other processes, while poorer processes are given the opportunity

²This is especially true for disk caches.

```
TWO-PROCESS PARTITIONING ALGORITHM
/* ShadowDirectoryA      :the shadow directory of Process A. */
/* PartitionedCache       :the cache containing the blocks of */
/*                           Process A and Process B. */
/* SmoothedIndexA(B)      :the array containing the sorted */
/*                           hit index distributions of A (B) */
/* PartitionA(B)           :the current amount of blocks of */
/*                           PartitionedCache allocated to A (B) */
/* HitIndexA(B)            :the array containing the number of */
/*                           hits recorded at all possible indices */
/*                           in the shadow directory of A (B). */
/* tau                     :size of the time window adopted by the */
/*                           algorithm. */

/* main_loop: */
do {
    get next block_address
    number_references = number_references + 1
    /* Assume block_address is from Process A. */
    update ShadowDirectoryA /* reorder lines in cache directory A
                               Update hit index distribution if
                               access results in hit */

    find block_address in PartitionedCache
    if (MISS) {
        if (SmoothedIndexA[PartitionA]
            > SmoothedIndexB[PartitionB]) {
            partitionA := partitionA + 1
            partitionB := partitionB - 1
            Update PartitionedCache by flushing LRU B line
        }
        else
            Update PartitionedCache by flushing LRU A line
    }
    else { /* HIT */
        Update PartitionedCache
        Update HitIndexA
    }

    if (number_references % TAU == 0) {
        sort HitIndexA into SmoothedIndexA
        sort HitIndexB into SmoothedIndexB
        divide all entries in HitIndexA by Ctau
        divide all entries in HitIndexB by Ctau
    }
}
while (1);
```

to increase their partition on every miss. These upward and downward movements tend to distribute the processes' wealth equally, as we show in the next section with a trace-driven simulation of a fully associative cache partitioned among 13 processes.

III. TRACE-DRIVEN COMPARISON OF STANDARD LRU AND PARTITIONED LRU

In this section we present the results of trace-driven simulations of fully associative caches of size 4 megabytes to 32

megabytes. The simulations compare caches managed by a conventional LRU algorithm to the same size caches managed by our partitioning algorithm. Two different sets of simulations were produced. One set of data is for a cache with large blocks, each 40 kilobytes in length, that can hold a full track of a disk. The other set of data treats a cache with smaller blocks, in this case 4 kilobytes, which is the size of a single sector of a disk track.

The trace contains approximately six hours of activity of a large disk cache on an IBM 3090 computer system in a

commercial environment. The trace consists of 1.6 million Channel Command Word (CCW) instructions, each directed to one of 13 physical disks that share a disk cache. On the IBM/370 architecture each I/O operation is started when one of the processors issues a Start I/O instruction [8]. Each Start I/O instruction triggers an I/O channel, which responds by executing a list of CCW's. The list is generally one CCW long, but may contain several. The trace used in our simulation contained CCW lists of length 1 exclusively, except for one list containing 2 CCW's. Each I/O access is represented by a record containing, among other quantities, the disk address, the block address, and a tag identifying *read* and *write* accesses.

All runs started with an initial partition allocating 1/13th of the cache to each disk. The smoothing of the hit-indexes was performed every $\tau = 10000$ references. The update of the raw hit-indexes to reflect the aging of the information was performed in synchronization with the smoothing, by dividing the raw hit-index distribution by $C_\tau = 2$, which can be done in practice by shifting each register by one bit to the right.

For each simulation run, 14 directories (13 shadow directories and one directory for the nonpartitioned cache) were managed completely in true LRU mode.

These values of τ and C_τ have been chosen through an ad-hoc process, and different values of τ and C_τ may give better results than those presented here. The accurate selection of τ and C_τ depends on many factors, such as the speed of the controller implementing the partitioning algorithm, the number of processes involved, their average lengths, as well as their locality characteristics. The speed of the controller will directly affect in the amount of time required to smooth and update the histograms for each process, and will affect the choice of τ . The dynamics of the processes in terms of their locality of lack thereof may also influence the choice of τ and C_τ .

The results of the simulations appear in Table I. The table shows both the *overall hit-ratio*, computed as the total number of hits (read or write) divided by the total number of accesses, and the *read hit-ratio*, computed as the total number of read hits divided by the total number of accesses, for both the partitioned and the full LRU caches, after 1 200 000 references have been processed. The relative change in hit ratio is the relative improvement in hit ratio in changing from an LRU-replacement algorithm to partitioned replacement. The data produced for the overall hit-ratio were obtained by computing histograms of all hits, whereas the read hit-ratio data were obtained on a different set of runs in which the histograms recorded only the read hits and ignored the write hits. The latter set of runs thus optimized the partitions in a way that sets the read hit-ratio derivatives equal rather than by setting the overall hit-ratio derivatives equal. The effect of this change produces better read hit-ratios for all caches studied except for the 32-megabyte sector cache in which the read hit-ratio produced by the two methods was so close as to be essentially the same.

Figs. 6 and 7 show the variation in virtual time of read hit-ratios and overall hit-ratios for the track-cache simulations. Figs. 8 and 9 show the same data for sector caches.

Note that the partitioning algorithm produces hit ratios a few percent higher than the LRU hit ratios in all cases except

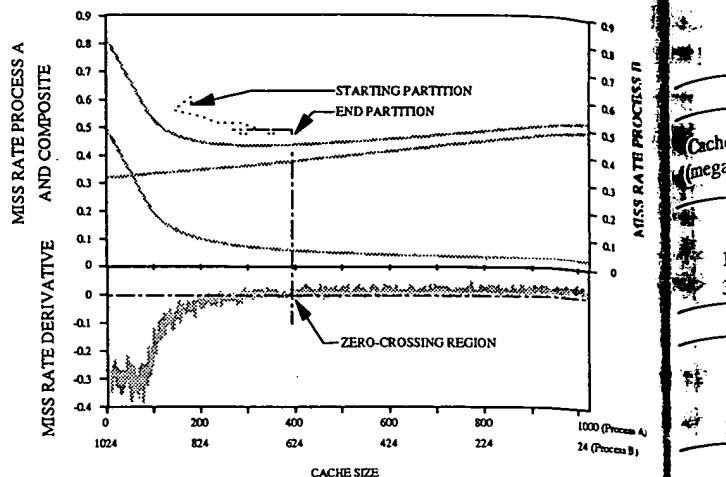


Fig. 5. Convergence of measured miss ratios.

for the 4-megabyte overall hit-ratio, which is only marginally different for the two replacement algorithms. The figures also reveal that the partitioning algorithm converges more quickly to the steady-state allocation, although the early values of the overall hit-ratios may briefly be higher for the LRU algorithm.

It is important to note that the hit-ratio shown in Figs. 6 and 7 shows the *cumulative* hit-ratios for the whole trace, i.e. the ratio of hits to the total number of references since the beginning of the trace. Hence, the difference in hit-ratios computed over a shorter window, toward the end of the trace, may be significantly higher than that of the cumulative hit-ratios shown in the figures.

Of all the cache sizes studied, the largest difference after steady state is reached (in the neighborhood of the 300 000th reference), occurs for the 8-megabyte cache and is roughly 2% in overall hit-ratio and 3% in read hit-ratio. Apparently, when a cache is large enough to capture the bulk of the possible hits, the replacement policy is not critical, and similarly, when a cache is too small to produce many of the possible hits, replacement policies are equally unsuccessful in capturing hits. Somewhere in the middle range of sizes, the cache is large enough for a replacement policy to exploit the memory available if it can produce good decisions for lines to replace.

To give another view of the simulation data, the distribution of read hits from the 32-megabyte sector cache simulation is plotted in Fig. 12. Of the 13 partitions, 12 can be seen. One of the 13 disks has an extremely low access frequency, which resulted in the algorithm allocating no cache lines to that disk. The partitions are shown side-by-side in the figure, and represent the state of the 8192-block cache after all 1.6 million I/O addresses are processed. The top part of the graph shows the hit-distribution in each partition. To verify that the partitions are allocated fairly to the 12 processes, we plotted in the lower part of Fig. 12 the value of the hit-index frequency at the index corresponding to the partition size. Since this value approximates the miss-rate derivative, it should be equal for all processes. Fig. 12 shows that this is the case. All the indexes have roughly the same amplitude, with the exception of the rightmost partitions, which happen to be the smallest ones. Two effects can account for amplitude variation visible in the

TABLE I
TRACK AND SECTOR CACHES: OVERALL AND READ HIT RATIOS

Cache Size (megabytes)	Track Cache				Read Hit-Ratio, Track			
	LRU (%)	Partit'd (%)	Difference Abs. (%)	Difference Rel. (%)	LRU (%)	Partit'd (%)	Difference Abs. (%)	Difference Rel. (%)
4	25.88	27.03	1.15	4.44	17.95	20.09	2.14	11.92
8	33.71	36.44	2.73	8.10	23.72	26.80	3.08	12.98
16	42.57	44.86	2.29	5.38	31.21	33.65	2.44	7.82
32	52.58	54.13	1.55	2.95	40.45	42.20	1.75	4.33
Sector Cache								
4	13.15	17.67	4.52	34.37	1.57	7.24	5.67	361.1
8	23.35	28.98	5.63	24.11	11.34	17.67	6.33	55.8
16	38.90	43.16	4.26	10.95	26.75	30.91	4.15	15.6
32	53.59	54.49	0.90	1.68	41.40	42.25	0.85	2.05

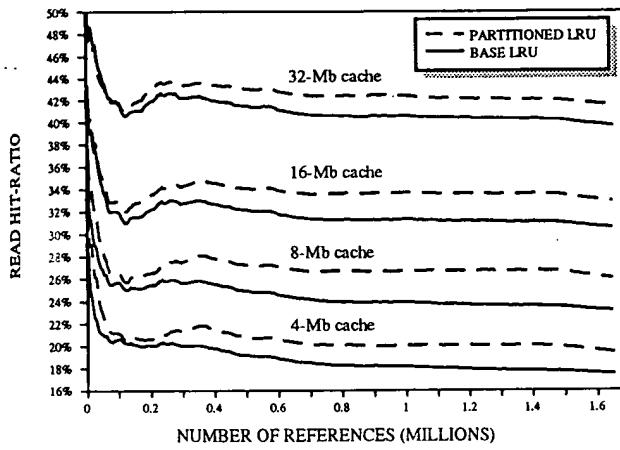


Fig. 6. Read hit-ratio for 4-megabyte to 32-megabyte track caches.

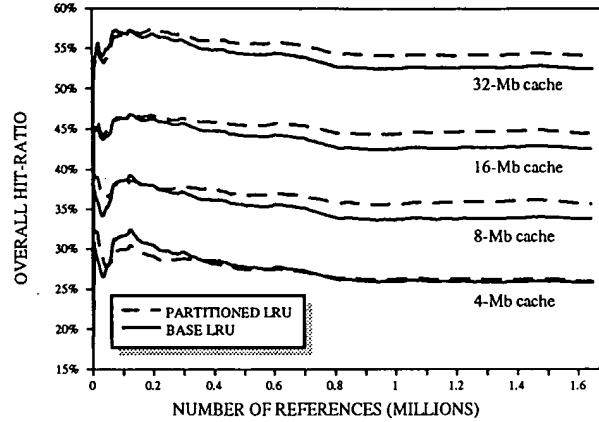


Fig. 7. Overall hit-ratios for 4-megabyte to 32-megabyte track caches.

figure. The first effect is that the smoothing process changes the estimate of the derivative rather rarely, and this estimate may, in fact, differ from the current histogram value when the time since the last smoothing has grown large. The second effect is that the smaller the partition is, the less faithful the smoothing process is to the raw distribution. Nonetheless, the partitioned algorithm still manages to allocate the available cache space fairly, and produces an improvement over the nonpartitioned LRU algorithm.

A. Influence of Hit-Ratio on Response-Time

Although the improvement in hit ratio achieved by the partitioning algorithm is small, the penalties imposed by misses are so high that even a small improvement in the hit ratio of a disk cache can be worthwhile.

To explore this issue further, we studied the influence of the disk cache hit-ratio on the disk system's response time. The disk-system studied is identical to that presented by Wolf in [18] and is shown in Fig. 10. An iterative queueing network model [18] is used to evaluate the average response time for different disk-access throughputs. The result is shown in Fig. 11(a) and (b). Fig. 11(a) shows the variation of the response time as a function of the disk cache hit-ratio for different CPU throughputs. For low CPU throughputs (lower than 100 accesses per second), the response time varies less with the hit-ratio. In such cases, the computer system is not

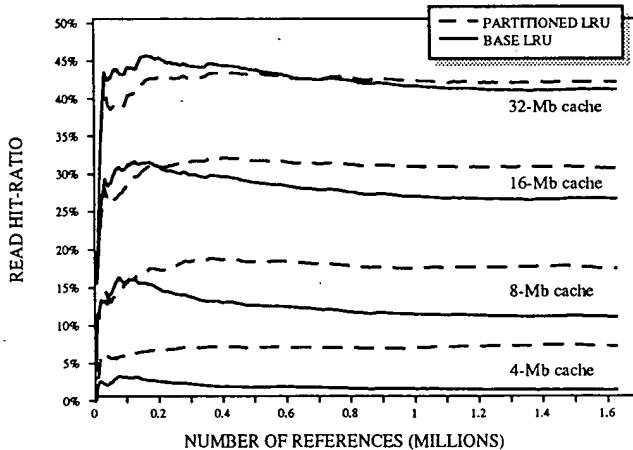


Fig. 8. Read hit-ratio for 4-megabyte to 32-megabyte sector caches.

utilized enough to gain dramatically from the addition of the disk cache. At throughputs of 300 accesses per second and higher, the response time varies significantly for small increases in hit-ratio. In those cases the I/O system is highly loaded, and small changes in hit-ratio have drastic effects on the system's performance. Fig. 11(b) confirms such system sensitivity by showing the average decrease in response time caused by increases in hit-ratio of 2.5%, 5.0%, 7.5%, and 10.0%, as a function of CPU throughput. For example, at a CPU throughput of 304 requests per second, an increase in hit-

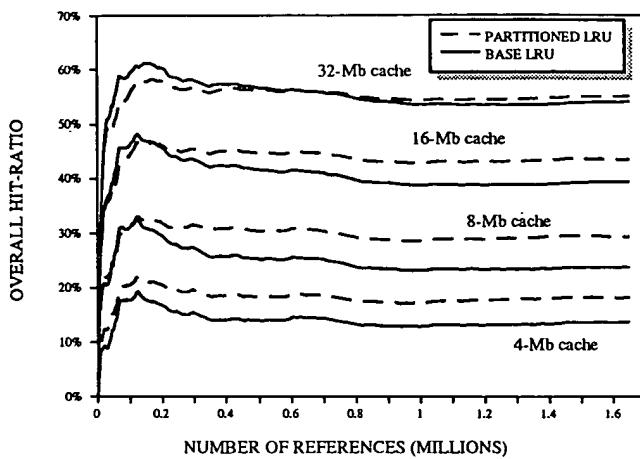


Fig. 9. Overall hit-ratio for 4-megabyte to 32-megabyte sector caches.

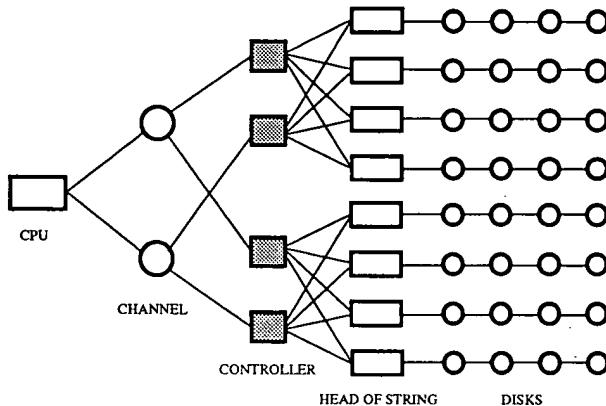


Fig. 10. Computer system configuration (disk cache located in controllers).

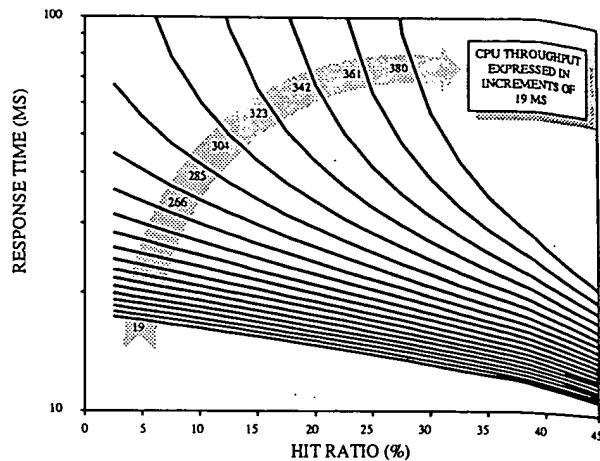
ratio of 2.5% will result in an average decrease in response time of 20.46 ms. Clearly, the partitioning algorithm will be most effective when the disk system is highly energized.

B. Tradeoff in Storage Allocation Between Data and Shadow Directory

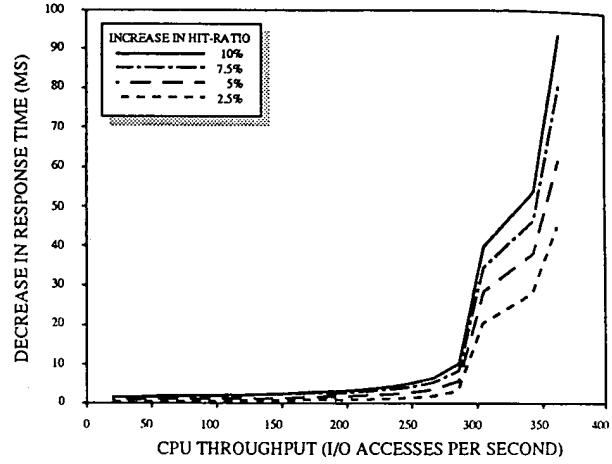
Also of interest is determining the tradeoff between improvement in hit ratio due to partitioning and improvement due to increase in cache size. The partitioning algorithm requires a shadow directory for each process competing for the cache, as well as two arrays to keep the raw and smoothed histograms of hit-indexes.

This directory must contain the block address of the track or sector whose copy is in the data storage of the cache. Since our traces contain 16-bit sector addresses, and since the track of the disk contains 10 sectors, the relative size increase for sector caches is 1.855%, and 0.186% for track caches, assuming that the cache is shared by 13 disks. The increase in hit-ratio due to partitioning is shown in Table II.

Table II compares the hit ratio for partitioned cache to the hit ratio of an augmented cache for which the storage used in shadow directories has been converted to data storage. The data for the augmented cache is computed by fitting the true hit-ratio curves as a function of cache size with a hyperbolic



(a)



(b)

Fig. 11. (a) Response time as a function of hit-ratio, for varying CPU throughputs. (b) Decrease in response time as a function of increase in hit ratio.

curve, and by interpolating the hit-ratio for the new sizes considered.

These values appear in the second column of the table. The hyperbolic fit for the track caches is $\text{Hit-ratio} = 4.63 \cdot 10^{-4} C^{0.39125}$, and $\text{Hit-ratio} = 1.512 \cdot 10^{-12} C^{1.54005}$ for sector caches, where C is the cache size. The last two columns show the absolute and relative increase in hit-rates.

Table II shows that storage is better allocated for the shadow directories than for storing the data. Sector caches appear to be the most likely caches to benefit from cache size increase.

IV. EXTENSION OF THE MODEL

In applications where the system performance can be expressed as a function of the hit ratio and/or of other measurable parameters that are affected by the replacement algorithm, one may be able to generalize the partitioning scheme of this paper to the more complex problem. For the sake of simplicity, let us denote the function representing the performance measure of a given process in a cache of size C as $f(C)$, and let us assume that $f(C)$ is a nonincreasing function of the cache size. The theory developed by Stone, Wolf, and Turek, indicates that the partition that maximizes performance allocates cache to

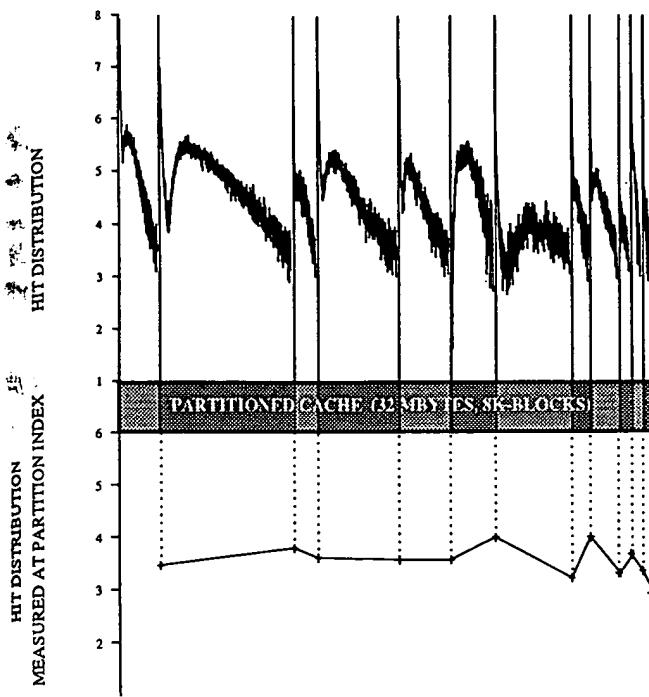


Fig. 12. Hit distributions of partitions of a sector cache.

TABLE II
TRADEOFF BETWEEN CACHE INCREASE AND PARTITIONING.
THE PARTITION IS UPDATED ON READ AND WRITE HITS

Cache Size (bytes)	Track Cache			
	Read Hit-Ratio LRU (%) (fitted)	Read Hit-Ratio Partition (%) (measured)	Absolute difference (%)	Relative difference (%)
4M	18.04	20.09	2.05	11.38
8M	23.66	26.80	3.14	13.26
16M	31.03	33.65	2.62	8.44
Sector Cache				
4M	2.46	7.24	4.78	194.53
8M	7.15	17.67	10.52	147.19
16M	20.79	30.91	10.12	48.69

each physical disk in a way that equates the performance-derivatives with respect to cache size for each physical cache. For example, performance may be slightly higher for records located in central tracks as opposed to records located on outer tracks. The performance derivative is greater for a record located on an outer track than one located on a central track, and thus a replacement algorithm should favor retention of records that reside on outer tracks over those on central tracks in order to produce equal performance-derivatives with respect to cache size.

V. SUMMARY AND CONCLUSIONS

The main result of this paper is a study that shows that an implementation of the Stone, Wolf, Turek cache-partitioning algorithm can produce a small relative improvement in disk-cache performance. The study confirms that LRU replacement does not produce an optimum partition of disk-cache memory,

but, in fact, it produces a partition that is very close to optimum.

For the data presented in this paper, the hit-ratio improvement available from the partitioning policy is fairly small, but it appears that the extra implementation cost of the partitioning algorithm is justified in systems where the throughput of CPU accesses to the disk is fairly high.

An interesting characteristic of the cache-partitioning algorithm is that it partitions the cache among competing processes by taking storage away from the process whose loss of cache memory causes the smallest decrease in performance. The nontrivial aspect of the implementation is to be able to estimate in real time the change in performance caused by a change in cache size. The paper uses a shadow directory of a large cache to hold a histogram of cache hits in order to produce the required data for driving the partitioning algorithm. The algorithm described here uses sorting to smooth the histograms, but other techniques may work equally well. To be able to track the change in process behavior over time, the algorithm estimates the frequency weighted miss-rate derivatives at discrete intervals of time, and then uses an aging scheme to reduce the influence of older behavior patterns.

Several research questions still remain open. The algorithm requires mechanisms for smoothing distributions and for aging history. Although the mechanisms used in the paper appear to be effective, other alternatives are worthy of investigation. Preliminary studies suggest that the algorithm is sensitive to the window between smoothing of the histograms, and that the increase in hit ratio depends somewhat on the window size. We conjecture that, for a given process, an optimal window exists such that the hit ratio obtained with this window is larger than with any other window values. Intuitively, when the window is infinitely large, the long past behavior of the process is as influential as is the near and most recent past. When the characteristics of the access pattern generated by a process change abruptly, the infinite window scheme will not record such changes until much later, and will react too slowly. On the other hand, if the window is very short, the algorithm may not have time to create a picture of the process's access pattern, and may not generate the best partition.

The partitioning algorithm works best when references from processes are interlaced among each other. If references tend to occur in a way such that strings of references from individual processes are not strongly interlaced in the composite reference string, there will be a tendency for each process to accumulate more cache than it deserves before the end of its string of references. It is an interesting open question to find a replacement algorithm that works well in this situation.

Another open problem concerns the management of cache when some processes are known to make sequential accesses. The algorithm in this paper does not treat sequential accesses as a special case, whereas, for some disk systems, the fact that a file is being read or written sequentially can be made known to a cache controller. What partitioning and replacement policies should be used when the cache controller can distinguish between sequential and nonsequential accesses?

In the final analysis, because LRU is so close to optimal, a cache-replacement algorithm that proves to be worthy of

implementation is very likely to be heavily based toward LRU replacement, with the deviations from LRU dependent on measured statistics. The algorithm presented in this paper is of this type.

ACKNOWLEDGMENT

The authors would like to thank K. Smith of IBM for providing the disk traces as well as greatly appreciated technical expertise.

REFERENCES

- [1] L. A. Belady and C. J. Kuhner, "Dynamic space sharing in computer systems," *Commun. ACM*, vol. 12, pp. 269-282, 1969.
- [2] J. L. Bentley and C. McGeoch, "Worst-case analysis of self-organizing sequential search heuristics," in *Proc. of 20th Allerton Conf. Commun., Contr., Comput.*, Univ. of Illinois, Urbana-Champaign, Oct. 6-8, 1982, pp. 452-461, 1983.
- [3] A. Brandwajn and J. Hernandez, "A study of a mechanism for controlling multiprogrammed memory in an interactive system," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 321-331, May 1981.
- [4] D. D. Chamberlin, S. H. Fuller, and L. Y. Liu, "An analysis of page allocation strategies for multiprogramming systems with virtual memory," *IBM J. Res. Develop.*, pp. 403-411, Sept. 1973.
- [5] W. W. Chu and H. Opderbeck, "The page fault frequency replacement algorithm," in *AFIPS Conf. Proc. Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 597-608.
- [6] P. Denning, K. C. Kahn, J. Leroudier, D. Potier, and R. Suri, "Optimal multiprogramming," *Acta Informatica*, vol. 7, pp. 197-216, 1976.
- [7] T. Ibaraki and N. Katoh, *Resource Allocation Problems*. Cambridge, MA: MIT Press, 1988.
- [8] IBM Corp., *IBM System/370: Principles of Operation*, GA22-7000-9, May 1983.
- [9] R. Fagin, "Asymptotic miss ratio over independent references," *J. Comput. Syst. Sci.*, vol. 14, pp. 222-250, 1977.
- [10] M. Z. Ghanem, "Dynamic partitioning of the main memory using the working set concept," *IBM J. Res. Develop.*, pp. 445-450, Sept. 1975.
- [11] D. Knuth, *The Art of Computer Programming*, Vol. 3. Reading, MA: Addison-Wesley, 1973.
- [12] T. Puzak, "An analysis of cache replacement algorithms," Ph.D. dissertation., Univ. of Massachusetts, Feb. 1985.
- [13] A. J. Smith, "Cache memories," *Comput. Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.
- [14] ———, "Disk cache: Miss ratio analysis and design considerations," *ACM Trans. Comput. Sys.*, vol. 3, no. 3, pp. 161-203, Aug. 1985.
- [15] H. S. Stone, J. L. Wolf, and J. Turek, "Optimal partitioning of cache memory," IBM Res. Rep. RC14444(#64697), pp. 1-25. Mar. 1989.
- [16] D. Thiébaut, "Influence of program transients in computer cache memories," Ph.D. dissertation, Univ. Massachusetts, Feb. 1988.
- [17] D. Thiébaut, H. S. Stone, and J. L. Wolf, "Synthetic traces for trace-driven simulation of cache memories," IBM Res. Rep. RC14268(#63748), Dec. 1988, pp. 1-49.
- [18] J. L. Wolf, "The placement optimization program: A practical solution to the disk file assignment problem," in *Proc. Sigmetrics Conf.*, Berkeley, CA, 1989.
- [19] J. L. Wolf, B. R. Iyer, K. R. Pattipati, and J. Turek, "Optimal buffer partitioning for the nested block join algorithm," in *Proc. Seventh Int. Conf. Data Eng.*, Kobe, Japan, Apr. 1991, pp. 510-519.
- [20] J. L. Wolf, D. Dias, and P. S. Yu, "An effective algorithm for parallelizing sort merge joins in the presence of data skew," in *Proc. Second Int. Conf. Databases in Parallel and Distributed Syst.*, Dublin, Ireland, July 1990, pp. 103-115.
- [21] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Reading, MA: Addison-Wesley, 1949.
- [22] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202-208, Feb. 1985.

Dominique Thiébaut (S'81-M'88), for a photograph and biography, see the April 1992 issue of this TRANSACTIONS, p. 410

Harold S. Stone (S'61-M'63-SM'81-F'87), for a photograph and biography, see the April 1992 issue of this TRANSACTIONS, p. 410

Joel L. Wolf, for a photograph and biography, see the April 1992 issue of this TRANSACTIONS, p. 410.

Abstract—
machines. For
size of data fi
the number
addresses, 3)
memory, and
these principle
A bypass
replacement t
performance
level simulate
the instructio
words achiev
compensates i
the extent tha
be integrated

Index Term
memory, bypass
placement, in
cache, perfor

DATA flc
puter s
synchronizat
cessing envi
important co
not yet been

A cache i:
that is, only
be satisfied
Conventio
ture usually
the necessar
temporal [1]

Thoreson
patterns in d
and tempor
et al. [3] pr
flow machin
the principle
combination:
however, as:
consists of c
and transfer
any spatial l
priately setti
time can be

Manuscript r
The author i
Laboratories, 3
IEEE Log N